

Progettazione di Sistemi Distribuiti e Big Data

Primo Homework

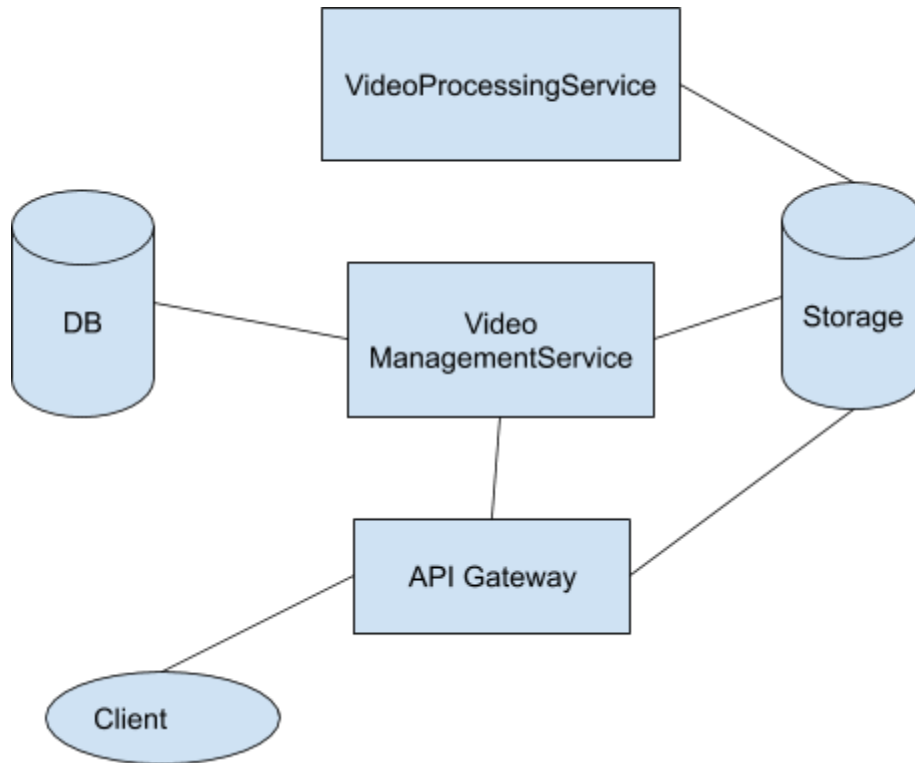
2019/2020

Info generali

- Tutti i progetti vanno sviluppati usando Git (e una piattaforma pubblica a scelta tra Github e Gitlab)
- Ogni componente dei progetti va sviluppato in un container docker separato
- Vanno sviluppati sia gli ambienti di sviluppo che gli ambienti di produzione (docker-compose)
- I client vanno, nel possibile, configurati come container docker separandoli dal punto di vista della rete dal servizio lato server
- Sviluppare le interfacce REST e le altre specifiche date di seguito rimanendo piu' fedeli possibile a quanto richiesto
- A tutti i progetti va allegata una relazione dettagliata delle scelte implementative e progettuali: puo' essere fornita in pdf o in markdown (README.MD nel repository git)
- Orario di ricevimento: (Meglio scrivere una mail precedentemente)
 - Lunedì 15.00-17.00
 - Mail: a.dimaria@studium.unict.it
 - Skype: andreadimaria400
 - Mercoledì 15.00-17.00
 - Mail: alessandro.distefano@phd.unict.it
 - Skype: aleskandrox

(a) Video Server

Implementare un sistema di gestione per video in formato MPEG-DASH.



Il componente **VideoManagementService** deve fornire una interfaccia REST con i seguenti endpoint e comportamenti associati:

- 1) POST /videos [Necessita autenticazione]
 - a) Prende in ingresso un JSON payload:
 - i) { "name": "\$VIDEO_NAME", "author": "\$AUTHOR_NAME" }
 - b) Salva nel database un record con i dati del payload. Il record deve avere un riferimento all'utente che lo ha creato
- 2) POST /videos/:id [Necessita autenticazione]
 - a) Prende in ingresso un file video mp4
 - b) Verifica l'esistenza del record con id :id nel database (Risponde con 400 se non esiste) e che il record sia associato all'utente autenticato
 - c) Salva il video sullo storage al path /var/video/:id/video.mp4
 - d) Manda una richiesta REST POST /videos/process al componente VideoProcessingService con payload json:
 - i) { "videoid": ":id" }

- e) Aggiorna il record :id nel database impostando lo stato su "uploaded"
- 3) GET /videos
 - a) Ritorna una lista in json dei video disponibili interrogando il database
- 4) GET /videos/:id
 - a) Verifica l'esistenza del record con id :id nel database (Risponde con 404 se non esiste)
 - b) Risponde con un HTTP Status code 301 puntando alla url /videofiles/:id/video.mpd
- 5) POST /register registra un utente per l'autenticazione e l'accesso alle chiamate POST.

Il componente **VideoProcessingService** espone una REST API su /videos/process che riceve un payload JSON come al punto 2.d.i del componente VideoManagementService e lancia un thread per eseguire l'encoding del video caricato tramite lo script bash allegato che fa uso di ffmpeg.

L'API Gateway e' configurato per:

- Inoltare tutte le richieste su /vms/* verso il componente VideoManagementService
- Inoltare tutte le richieste su /videofiles sulla rootdir /var/videofiles
- Configurare opportunamente l'API Gateway in termini di timeout massimi e dimensione dei file massima in upload

Info su VideoProcessingService

All'arrivo della richiesta va lanciato un thread che esegua uno script come il seguente

**ffmpeg -i \$inputFile **

**-map 0:v:0 -map 0:a?:0 -map 0:v:0 -map 0:a?:0 -map 0:v:0 -map 0:a?:0 -map 0:v:0 -map 0:a?:0 -map 0:v:0 -map 0:a?:0 -map 0:v:0 -map 0:a?:0 **

**-b:v:0 350k -c:v:0 libx264 -filter:v:0 "scale=320:-1" **

**-b:v:1 1000k -c:v:1 libx264 -filter:v:1 "scale=640:-1" **

**-b:v:2 3000k -c:v:2 libx264 -filter:v:2 "scale=1280:-1" **

**-b:v:3 245k -c:v:3 libvpx-vp9 -filter:v:3 "scale=320:-1" **

**-b:v:4 700k -c:v:4 libvpx-vp9 -filter:v:4 "scale=640:-1" **

**-b:v:5 2100k -c:v:5 libvpx-vp9 -filter:v:5 "scale=1280:-1" **

**-use_timeline 1 -use_template 1 -window_size 6 -adaptation_sets "id=0,streams=v id=1,streams=a" **

-hls_playlist true -f dash output/path/video.mpd

<https://stackoverflow.com/questions/48256686/how-to-create-multi-bit-rate-dash-content-using-ffmpeg-dash-muxer>

Alternativamente (meno consigliato):

<https://blog.infireal.com/2018/04/mpeg-dash-with-only-ffmpeg/>

Consiglio: testare lo script per capirne la gestione dei path e quant'altro di necessario nel proprio host con un video a caso (corto) e manualmente da terminale prima di implementare il servizio VideoProcessingService che lancia i job

Consiglio 2: nei test usare principalmente video di durata massima di 1 minuto.

Varianti

DB1: Utilizzare database MySql

DB2: Utilizzare database Mongo

GW1: Implementare API Gateway attraverso Nginx

GW2: Implementare API Gateway come una applicazione SpringBoot (Spring Cloud Gateway)

STATS1:

Configurare l'API Gateway per fornire statistiche su:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Eventuali errori
- Richieste al secondo

Valutare il mezzo su cui conservare i dati (e.g. FileSystem, Database, Prometheus....)

STATS2:

Implementare un client che esegua:

Load1:

- n1 POST / che creino n1 nuovi record
- n1 POST /:id (per ogni id ricevuto) che eseguano l'upload di un video casuale (puo' sempre essere lo stesso)

Load2:

- n2 GET /

Load3:

- n3 GET /:id (casualmente scegliere un video esistente con probabilita' p1, e un video non esistente con probabilita' 1 - p1)

Per ogni richiesta inviata esportare in csv i seguenti dati:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Codice di ritorno (200 o codice di error)

Esempi di valori per i parametri del load: $n1 = 10$, $n2 = 100$, $n3 = 50$, $p1 = 0.9$

Consiglio: utilizzare variabili d'ambiente o argomenti da riga di comando (argv) per gestire i parametri esposti sopra.

Puo' essere implementato con un bash/python script che faccia uso di un client cli per dash o con node.js se si preferisce utilizzare dash.js

Client utilizzabili su command-line: ffplay, MP4Client

Client utilizzabile js-based: dash.js

<http://ronallo.com/blog/testing-dash-and-hls-streams-on-linux/>

<https://github.com/Dash-Industry-Forum/dash.js/wiki>

STATS3:

Configurare i database per fornire statistiche sulle query e lo stato del container in cui esegue:

- Tipo di query
- Tempo richiesto per l'esecuzione
- Eventuali errori
- Richieste al secondo
- Risorse occupate
- Payload size input
- Payload size output

Valutare il mezzo su cui conservare i dati di monitoraggio (e.g. Ffilesystem, Database, Prometheus...)

STATS4:

L'API Gateway aggiunge un header http custom "X-REQUEST-ID" che deve essere propagato per ogni richiesta su tutti i componenti che interagiscono (no database). Il valore di questo header deve essere univoco per ogni request (utilizzare per esempio il timestamp UNIX di arrivo della richiesta all'API Gateway)

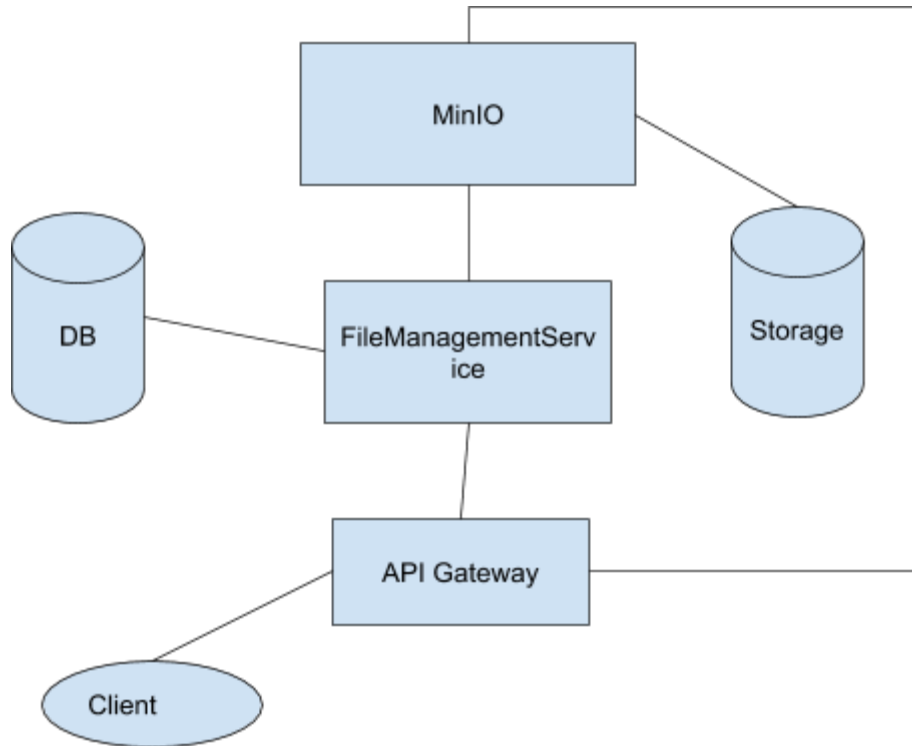
Per ogni componente (escluso l'API Gateway) implementare i metodi in modo da conservare nel database un record per ogni chiamata che riporti:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Codice di ritorno (200 o codice di error)
- Valore X-REQUEST-ID

- Nome componente che scrive nel db (consiglio, usare hostname del container in cui esegue)

(b) Object Storage Adapter

Realizzare un sistema distribuito che implementi un **adapter per sistemi di object storage.**



Il componente FileManagementService fornisce una API Rest che risponde in questo modo:

1) GET /:id [Necessita autenticazione]

- a) Se :id non esiste nel database oppure l'utente autenticato ha ruolo USER e non è l'owner del file risponde con 404
- b) Ritorna un http 301 che punti alla url a cui poter trovare il file:
/minio/\$generatedMinIOURI

2) GET / [Necessita autenticazione]

- DONE** a) Ritorna in Json una lista dei file disponibili di cui l'utente autenticato è owner
- b) Se l'utente autenticato ha ruolo ADMIN ritorna in Json una lista di tutti i file disponibili

3) POST / [Necessita autenticazione]

- DONE** a) Consuma un json con { fileName: \$fileName, author: \$author }
- b) Salva il dato ricevuto e il riferimento all'utente come un nuovo record nel database

- c) Risponde con 200 e il json object che rappresenta il nuovo record creato (incluso id)
- 4) POST /:id [Necessita autenticazione]
 - a) Verifica che :id sia già stato creato nel database e che il record sia associato all'utente autenticato (se no risponde con 400)
 - b) Consuma un file di qualunque tipo
 - DONE** c) Inoltra il file su MinIO in un bucket di default
 - d) Aggiorna il record del database con i dati ritornati da MinIO per il raggiungimento del file (in particolare objectName e bucket)
 - e) Risponde con l'oggetto finale del database che include l'id
- 5) DELETE /:id [Necessita autenticazione]
 - a) Elimina il file dallo storage MinIO e il record dal database
 - i) L'utente con ruolo USER può eliminare soltanto i file di cui è owner
 - ii) L'utente con ruolo ADMIN può eliminare anche i file degli altri utenti
 - b) Se il file non esisteva risponde con 404
- 6) POST /register
 - a) Consuma un Json con {nickname: \$nickname, password: \$password, email: \$email }
 - DONE** b) Registra l'utente assegnandogli il ruolo di "USER" o "ADMIN" per permettergli l'autenticazione successiva per i metodi POST ai punti 3,4 e DELETE al punto 5

L'API Gateway e' configurato per:

- Inoltare tutte le richieste su /fms/* verso il componente FileManagementService
- Inoltare tutte le richieste su /minio verso il componente MinIO (la url generata da minIO attraverso presignedGetObject(...) dovra' essere modificata opportunamente per essere raggiungibile dall'esterno attraverso il reverse proxy configurato nell'API Gateway
- Configurare opportunamente l'API Gateway in termini di timeout massimi e dimensione dei file massima in uploads
- Ad esempio l'url restituita dalla chiamata presignedGetObject(...) restituisce l'url [http://miniourl:minioport/\\$URI](http://miniourl:minioport/$URI). Questa deve diventare [http://\\$apigatewaypublicurl:\\$apigatewaypublicport/minio/\\$URI](http://$apigatewaypublicurl:$apigatewaypublicport/minio/$URI) (\$apigatewaypublicurl potrebbe essere localhost)

Info su MinIO

Informazioni su container docker e quickstart:

<https://docs.min.io/docs/minio-quickstart-guide.html>

Autenticarsi con MinIO dal servizio adapter con chiave unica.

Utilizzare un solo bucket di default.

Si puo' interagire con MinIO in diverse modalita'.

Esempio 1 (strada meno convenzionale): usare il client unix mc da riga di comando (comporta l'utilizzo di "fork" nel codice per eseguire il comando necessario).

Esempio 2: usare MinIO Java SDK, disponibile su Maven, <https://github.com/minio/minio-java>

<https://golb.hplar.ch/2017/02/Upload-files-from-Java-to-a-Minio-server.html>

<https://docs.min.io/docs/java-client-api-reference.html#presignedGetObject>

Opzionale: usare un bucket diverso per ogni utente.

Varianti

DB1: Utilizzare database MySql

DB2: Utilizzare database Mongo

GW1: Implementare API Gateway attraverso Nginx

GW2: Implementare API Gateway come una applicazione SpringBoot (Spring Cloud Gateway)

STATS1:

Configurare l'API Gateway per fornire statistiche su:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Eventuali errori
- Richieste al secondo

Valutare il mezzo su cui conservare i dati (e.g. FileSystem, Database, Prometheus....)

STATS2: test

Implementare un client che esegua:

Load1:

- n1 POST / che creino n1 nuovi record
- n1 POST /:id (per ogni id ricevuto) che eseguano l'upload di un file casuale (puo' sempre essere lo stesso)

Load2:

- n2 GET / al secondo

Load3:

- n3 GET /:id al secondo (casualmente scegliere un video esistente con probabilita' p1, e un file non esistente con probabilita' 1 - p1)

Load4:

- n4 DELETE /:id (casualmente scegliere un video esistente con probabilita' p2, e un file non esistente con probabilita' 1 - p2)

-

Per ogni richiesta inviata esportare in csv i seguenti dati:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Codice di ritorno (200 o codice di error)

Esempi di valori per i parametri del load: $n1 = 10$, $n2 = 100$, $n3 = 50$, $n4 = 20$, $p1 = 0.9$, $p2 = 0.87$

Consiglio: utilizzare variabili d'ambiente o argomenti da riga di comando (argv[]) per gestire i parametri esposti sopra.

Il client può essere uno bash/python script. Se si usa bash fare riferimento a curl, con python esistono diverse librerie come "request" per effettuare http requests.

STATS3:

Configurare i database per fornire statistiche sulle query e lo stato del container in cui esegue:

- Tipo di query
- Tempo richiesto per l'esecuzione
- Eventuali errori
- Richieste al secondo
- Risorse occupate
- Payload size input
- Payload size output

Valutare quali delle precedenti statistiche possano essere implementate.

Valutare il mezzo su cui conservare i dati di monitoraggio (e.g. Filesystem, Database, Prometheus...)

STATS4:

L'API Gateway aggiunge un header http custom "X-REQUEST-ID" che deve essere propagato per ogni richiesta su tutti i componenti che interagiscono (no database). Il valore di questo header deve essere univoco per ogni request (utilizzare per esempio il timestamp UNIX di arrivo della richiesta all'API Gateway)

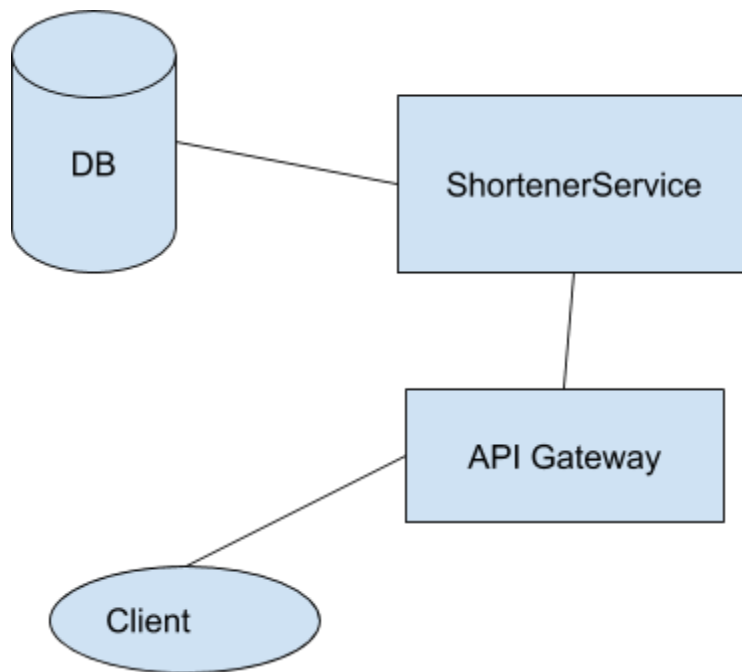
Per ogni componente (escluso l'API Gateway) implementare i metodi in modo da conservare nel database un record per ogni chiamata che riporti:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Codice di ritorno (200 o codice di error)
- Valore X-REQUEST-ID
- Nome componente che scrive nel db (consiglio, usare hostname del container in cui esegue)

(c) URL Shortener (Singolo)

Progettare un sistema di URL shortening che permetta, dato un URL (ipoteticamente lungo) di fornire un URL corto che faccia da redirect verso l'URL di input.

Esempi di sistemi simili: Tinyurl.com, Bit.ly



Il componente ShortenerService fornisce una API Rest che risponde in questo modo:

- 1) GET /:hash
 - a) Cerca nel db un record che abbia il campo alias uguale a quello fornito (:hash), se non esiste nel database risponde con 404
 - b) Ritorna un http 301 che punti alla url originale
- 2) POST /
 - a) Consuma un json con { uri: \$destinationUrl, [Optional] alias: \$alias }
 - b) Verifica, se presente che l'alias non sia già stata utilizzato per identificare una url
 - c) Se il custom alias non è presente generarne uno di lunghezza fissa che sia univoco rispetto agli altri già presenti nel db (vedere <https://tinyurl.com/ujlk88c> paragrafo 6)
 - d) Salva il dato ricevuto come un nuovo record nel database (usare il custom alias se possibile (presente e disponibile) o l'alias generato automaticamente)
 - e) Risponde con 200 e il json object che rappresenta il nuovo record creato (incluso id e shortUrl completa)

Varianti

DB1: Utilizzare database MySql

DB2: Utilizzare database Mongo

GW1: Implementare API Gateway attraverso Nginx

GW2: Implementare API Gateway come una applicazione SpringBoot (Spring Cloud Gateway)

STATS1:

Configurare l'API Gateway per fornire statistiche su:

- API richiesta (GET POST PUT ecc + URI)
- Payload size input
- Payload size output
- Tempi di risposta
- Eventuali errori
- Richieste al secondo

Valutare il mezzo su cui conservare i dati (e.g. FileSystem, Database, Prometheus....).

Progetti

PR1) (a).DB1.GW1.STATS1
PR2) (a).DB1.GW1.STATS2
PR3) (a).DB1.GW1.STATS3
PR4) (a).DB1.GW1.STATS4
PR5) (a).DB1.GW2.STATS1
PR6) (a).DB1.GW2.STATS2
PR7) (a).DB1.GW2.STATS3
PR8) (a).DB1.GW2.STATS4
PR9) (a).DB2.GW1.STATS1
PR10) (a).DB2.GW1.STATS2
PR11) (a).DB2.GW1.STATS3
PR12) (a).DB2.GW1.STATS4
PR13) (a).DB2.GW2.STATS1
PR14) (a).DB2.GW2.STATS2
PR15) (a).DB2.GW2.STATS3
PR16) (a).DB2.GW2.STATS4
PR17) (b).DB1.GW1.STATS1
PR18) (b).DB1.GW1.STATS2
PR19) (b).DB1.GW1.STATS3
PR20) (b).DB1.GW1.STATS4
PR21) (b).DB1.GW2.STATS1
PR22) (b).DB1.GW2.STATS2
PR23) (b).DB1.GW2.STATS3
PR24) (b).DB1.GW2.STATS4
PR25) (b).DB2.GW1.STATS1
PR26) (b).DB2.GW1.STATS2
PR27) (b).DB2.GW1.STATS3
PR28) (b).DB2.GW1.STATS4
PR29) (b).DB2.GW2.STATS1
PR30) (b).DB2.GW2.STATS2
PR31) (b).DB2.GW2.STATS3
PR32) (b).DB2.GW2.STATS4
PR33) (c).DB1.GW1.STATS1 (SINGOLO)
PR34) (c).DB1.GW2.STATS1 (SINGOLO)
PR35) (c).DB2.GW1.STATS1 (SINGOLO)